Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures

Joachim Niehren, David Sabel, Manfred Schmidt-Schauss, Jan Schwinghammer

Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures

Joachim Niehren[a] David Sabel[b] Manfred Schmidt-Schauß[b]
Jan Schwinghammer[c]

[a] *INRIA Futurs, Lille, France, Mostrare Project*
[b] *J. W. Goethe-Universität, Frankfurt, Germany*
[c] *Saarland University, Programming Systems Lab, Saarbrücken, Germany*

Abstract

We present an observational semantics for $\lambda(\mathsf{fut})$, a concurrent λ-calculus with reference cells and futures. The calculus $\lambda(\mathsf{fut})$ models the operational semantics of the concurrent higher-order programming language Alice ML. Our result is a powerful notion of equivalence that is the coarsest nontrivial congruence distinguishing observably different processes. It justifies a maximal set of correct program transformations, and it includes all of $\lambda(\mathsf{fut})$'s deterministic reduction rules, in particular, call-by-value β-reduction.

Keywords: Concurrent programming languages, lambda calculus, semantics

1 Introduction

The λ calculus $\lambda(\mathsf{fut})$ [12] models the operational semantics of the core of Alice ML [16], a recent concurrent higher-order programming language of the ML family [10,3] inspired by Mozart-Oz [20]. Alice ML provides typed functional programming with mixed concurrent eager and lazy threads which may be distributed transparently over the network. In Alice ML, futures form a (light-weight) concurrency primitive, and implement lazy loading of program components [15].

Formally, $\lambda(\mathsf{fut})$ is a call-by-value λ-calculus with reference cells and concurrent threads that synchronize on futures. Futures are like logic variables with restricted read and write access. Successful threads evaluate expressions to values and bind them to futures. Threads may be started eagerly or lazily and then operate call-by-value. Some reductions may proceed with futures as arguments while others require proper values (so they have to wait until the required futures get bound to proper values). This way, futures lead to a convenient form of (automatic) data-driven synchronization [1]. In contrast to previous purely functional λ-calculi with futures

[6], $\lambda(\mathsf{fut})$'s reference cells permit to express synchronization constructs, such as channels of the asynchronous π-calculus [9], JoCaml-like joins [4], and streams.

In this paper we present an observational semantics for $\lambda(\mathsf{fut})$ that is needed for reasoning about concurrent programs and program transformations. The result is a powerful notion of program equivalence that is the coarsest congruence distinguishing observably different processes. Thus, the congruence justifies a maximal set of correct program transformations, and we prove that in particular it includes call-by-value β-reduction (but not general β-reduction). To the best of our knowledge, no previous semantics with these properties exists for concurrent higher-order languages. Alternative approaches using encodings of $\lambda(\mathsf{fut})$ into other concurrent formalisms – such as (typed) π-calculi – that induce a semantics also for the source language, may work in principle. But then correctness of the encoding itself requires an additional justification, and unfortunately, we do not know how to prove that observational semantics is retained, so we prefer to work directly on the language $\lambda(\mathsf{fut})$.

Our observational semantics treats processes that cannot avoid a final error situation or a loop as equivalent, and distinguishes them from all others. Based on the operational semantics of $\lambda(\mathsf{fut})$ we construct the observational semantics as follows. First of all, we replace call-by-value β-reduction by *sharing call-by-value β-reduction* where substitutions are performed explicitly and lazily, via $\lambda(\mathsf{fut})$'s machinery for future dereferencing. This sharing variant is indispensable for proofs, but call-by-value β-reduction is correct, as we show. Second, we define a notion of *successful termination* for $\lambda(\mathsf{fut})$ processes. It requires that all started threads are terminated, that there are no needed lazy threads, and that all futures of eager threads have been bound to proper values, possibly through several indirections. Since the calculus is non-deterministic, we introduce notions of *may- and must-convergence* in a third step, a modelling technique used previously for lambda-calculi with *amb* [2,11,17]. We call a process may-convergent if it may terminate successfully, and must-convergent, if all its reduction descendants must always remain may-convergent. This notion of must-convergence is appropriate with respect to fair reduction operational semantics (see also [2,17]), that we assume for a real implementation of an evaluator. Fourth, we call two processes *observably equivalent*, if they exhibit the same may- and must-convergence in all contexts.

The following results are obtained. The observational equivalence is a congruence that includes call-by-value β-reduction and all other deterministic rules of $\lambda(\mathsf{fut})$. A slightly unexpected result was that the reductions involving so-called handles (see Section 2) turn out to be correct transformations, hence they are also "deterministic". The only non-deterministic rule is value exchange for reference cells. We distinguish particular instances of cell exchange that preserve observational equivalence. We also show that "garbage collection" and "path compression" are correct transformations, and present a non-trivial example for optimization (Subsection 4.5). The main tools in proving equivalences, besides the context lemma, are *complete sets of forking* and *commuting diagrams* [8,17].

Related work. Notions of program equivalence for deterministic languages with reference cells based on contextual equivalence, as e.g. considered in [13], have to be extended and adapted to take non-determinism and concurrency into ac-

$$
\begin{aligned}
x, y, z &\in \mathit{Var} \\
c \in \mathit{Const} &::= \mathbf{unit} \mid \mathbf{cell} \mid \mathbf{thread} \mid \mathbf{handle} \mid \mathbf{lazy} \\
e \in \mathit{Exp} &::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{exch}(e_1, e_2) \\
v \in \mathit{Val} &::= x \mid c \mid \lambda x.e \\
p \in \mathit{Process} &::= p_1 \mid p_2 \mid (\nu x)p \mid x\,\mathsf{c}\,v \mid x \Leftarrow e \mid y\,\mathsf{h}\,x \mid y\,\mathsf{h}\,\bullet \mid x \overset{susp}{\Longleftarrow} e
\end{aligned}
$$

Fig. 1. Expressions and processes of $\lambda(\mathsf{fut})$

count. Contextual equivalence as a combination of may- and must-convergence is also known from the use of convex powerdomains in domain-theoretic models [14]. However, observational semantics provides a powerful and also practically useful semantics with a maximum of equations in the presence of lambda abstractions, concurrency and synchronization, dynamically created threads, state through reference cells, and sharing, a combination which is not handled by semantical models built upon powerdomains. Most previous work on semantics for concurrent languages focuses on process calculi [9,19] or investigates the theory obtained by encoding lambda calculi into process calculus (for instance, [18]). In [5,7], program behavior in fragments of Concurrent ML is characterized by bisimilarity with respect to a labelled transition system.

Plan. We recall $\lambda(\mathsf{fut})$ in Section 2 and adapt it by sharing call-by-value β-reduction. Section 3 introduces the notion of successful termination and defines the may-must observational equivalence. We prove a context lemma which shows that observations in evaluation contexts suffice to characterize program equivalence. In Section 4, we consider correctness of program transformations with respect to observational equivalence.

2 Lambda(Fut)

We recall an untyped version of $\lambda(\mathsf{fut})$ from [12] and adapt it by sharing call-by-value β-reduction. This change does not affect the observational semantics, which includes call-by-value β-reduction nevertheless, as we will prove in Section 4.

The syntax of $\lambda(\mathsf{fut})$ is defined in Fig. 1. It has two layers, standard λ-expressions $e \in \mathit{Exp}$ for sequential computation within threads, and processes $p \in \mathit{Process}$ composing threads in parallel. New operations in expressions are introduced by (higher-order) constants: **unit** is a dummy value, and constants **thread**, **lazy** and **handle** serve for introducing eager threads, lazy threads, and handles, each of them together with a future. The constant **cell** introduces reference cells, and $\mathbf{exch}(e_1, e_2)$ expresses atomic exchange of cell values. Values v are defined as usual in a call-by-value λ-calculus. Note that values subsume variables and thus futures, even though the latter are not proper values. The only variable binder in expressions is λ. The set of free variables of e is denoted by $\mathsf{fv}(e)$. We identify expressions up to consistent renaming of bound variables and write $e[e'/x]$ for the (capture-free) substitution of e' for x in e.

As in the π-calculus, processes p of $\lambda(\mathsf{fut})$ are composed from components by parallel composition $p_1 \mid p_2$ and new name creation $(\nu x)p$, the latter is a variable

$$p_1 \mid p_2 \;\equiv\; p_2 \mid p_1 \qquad (p_1 \mid p_2) \mid p_3 \;\equiv\; p_1 \mid (p_2 \mid p_3)$$
$$(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p \qquad (\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2) \qquad \text{if } x \notin \mathsf{fv}(p_2)$$

Fig. 2. Structural congruence of processes

binder. The set of free variables of p is denoted by $\mathsf{fv}(p)$. The usual *structural congruence* $\equiv$ on processes is defined by the axioms in Fig. 2. We distinguish five types of components (all different from the π-calculus): (eager concurrent) threads $x \Leftarrow e$ will eventually bind future x to the value of expression e unless it diverges or suspends; x is called a *concurrent future*. Lazy threads $x \overset{susp}{\Longleftarrow} e$ are suspended computations that will start once the proper value of x is needed elsewhere; we call x a lazy future. Cells $x \,\mathsf{c}\, v$ associate (memory location) x to a value v. Handle components $y \,\mathsf{h}\, x$ associate handles y to futures x, so that y can be used to assign a value to x. We call x a future handled by y, or more shortly a *handled future*. Finally, a used handle component $y \,\mathsf{h}\, \bullet$ indicates that y is a handle that has already been used to bind its future.

A process p *introduces a variable* x if p contains some component of the following form (for some y, e, v):

$x \Leftarrow e$	x *is concurrent future* (for e)	$x \overset{susp}{\Longleftarrow} e$	x *is lazy future* (for e)
$x \,\mathsf{c}\, v$	x *is cell* (with content v)	$x \,\mathsf{h}\, y$	x *is handle* (for future y)
$y \,\mathsf{h}\, x$	x *is handled future* (handled by y)	$x \,\mathsf{h}\, \bullet$	x *is used handle*

Introduced variables are also called *process variables*. A process is *well-formed* iff it does not introduce any variable twice.

The binding operator ν can be seen as defining the observational scope of variables. Using the distinct variable convention and moving ν-binders to the top-level, a process p that introduces variables $\{\overline{x}, \overline{y}\}$ can be written in the form $(\nu\overline{x})p$, where p does not contain further ν-binders. The variables in $\{\overline{y}\}$ are directly observable by an external observer. The others in $\{\overline{x}\}$ may still be observable indirectly.

The operational semantics defines an evaluation strategy ev via contexts in that reduction rules apply. A *context* is a process or an expression with exactly one occurrence of the hole marker, i.e. the special constant $[\,]$. The hole marker cannot occur at the positions that are reserved for variable introduction, and in a cell $x \,\mathsf{c}\, v$, the position of the hole can only be in e for $v = \lambda x.e$. Let γ be a context, and η be a term or a process that can be plugged into its hole, then we write $\gamma[\eta]$ for the result of replacing $[\,]$ in γ by η (possibly capturing free variables of η).

With C we denote any context that is a process. We call C *flat* if its hole does not occur below a λ-binder, and *deep* otherwise. A context D denotes a process-context, where the hole marker occurs in process position. In Fig. 3 we define particular flat contexts of type expression that we call *evaluation contexts* (ECs) E and *future ECs* F. ECs encode the standard call-by-value, left-to-right reduction strategy, while future ECs control dereferencing operations on futures and starting suspended threads: dereferencing a future or starting the corresponding suspended thread is only allowed when the future's value is needed for a thread to proceed.

We define the operational semantics of $\lambda(\mathsf{fut})$ using a (small-step) reduction denoted by $\rightarrow$, or $\overset{\mathsf{ev}}{\longrightarrow}$ in case we want to distinguish it from the general transformations

introduced in Section 4. It is the least binary relation on processes $p \to p'$ satisfying the rules in Fig. 4. Sharing call-by-value β-reduction (β-CBV$_{\mathrm{L}}$(ev)) replaces binders λy by (νy) and binds y to the function's argument. Writing *let* $x = e$ *in* e' instead of $(\lambda x.e')e$, sharing β-reduction takes the form well-known from calculi with explicit closures: $E[let\ x = v\ in\ e] \to (\nu x)(x{\Leftarrow}v \mid E[e])$. Rule (FUT.DEREF(ev)) replaces needed occurrences of x by v. Basically, this recovers standard call-by-value β-reduction, as proved in Theorem 4.23. For instance:

$$E[(\lambda x_1 \lambda x_2.x_1)\, z\, z'] \xrightarrow{\beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev})} \xrightarrow{\beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev})} (\nu x_1)(\nu x_2)(E[x_1] \mid x_1{\Leftarrow}z \mid x_2{\Leftarrow}z')$$
$$\xrightarrow{\text{FUT.DEREF}(\mathsf{ev})} \quad E[z] \mid (\nu x_1)(\nu x_2)(x_1{\Leftarrow}z \mid x_2{\Leftarrow}z') \sim E[z]$$

The final garbage collection step will be licensed by our observational equivalence (Theorem 4.8).

Rule (THREAD.NEW(ev)) spawns a new eager thread $x{\Leftarrow}e$, where x may occur in e, so it may be viewed as a recursive let $x = e$. Similarly, (LAZY.NEW(ev)) creates a new suspended computation $x \overset{susp}{\Longleftarrow} e$. Dereferencing of future values (FUT.DEREF(ev)) and triggering of suspended computations (LAZY.TRIGGER(ev)) is controlled by future evaluation contexts F. Rule (HANDLE.NEW(ev)) introduces handle components $y\,\mathsf{h}\,x$ with static scope in e; the application $x\,v$ in (HANDLE.BIND(ev)) "consumes" the handle x and binds y to v, resulting in a used handle $x\,\mathsf{h}\,\bullet$ and thread $x{\Leftarrow}v$. Rule (CELL.NEW(ev)) creates new cells $z\,\mathsf{c}\,v$ with contents v. The exchange operation $\mathbf{exch}(z, v_1)$ writes v_1 to the cell and returns the previous contents. Since this is an *atomic* operation, no other thread can interfere. Note also that reduction preserves well-formedness and non-well-formedness.

Depending on the context, we will write $\bot$ for both an arbitrary non-converging expression (such as $\mathbf{thread}\ \lambda x.x$) or process (such as $x{\Leftarrow}x\,x$) in the following.

Example 2.1 *We define a binary choice operator that non-deterministically selects one of its two arguments. First we define demonic-choice as an eager function, so that both alternatives will be evaluated before actually choosing. As usual let* $K_1 = \lambda x \lambda y.x$ *and* $K_2 = \lambda x \lambda y.y$.

$$\begin{aligned} \mathit{demonic\text{-}choice} =_{\mathit{def}} \lambda u_1.\lambda u_2.\ &\mathit{let}\ z = \mathbf{cell}\ K_1\ \mathit{in} \\ &\mathit{let}\ y = \mathbf{thread}\ (\lambda\_.\mathbf{exch}(z, K_2))\ \mathit{in}\ (\mathbf{exch}(z, K_2))\, u_1\, u_2 \end{aligned}$$

For arbitrary values v_1, v_2 *one can verify for* $i = 1, 2$ *that* $D[E[\mathit{demonic\text{-}choice}\ v_1\ v_2]]$ *may reduce to* $D[E[v_i]]$ *modulo garbage collection, so the respective other branch cannot become active. For all* e_1, e_2, *we define a lazy choice expression* $\mathit{choice}[e_1, e_2]$ *that first chooses between* e_1 *and* e_2 *and then evaluates only the chosen expression:*

$$\mathit{choice}[e_1, e_2] =_{\mathit{def}} \mathit{demonic\text{-}choice}\ (\mathbf{lazy}\ \lambda\_.e_1)\ (\mathbf{lazy}\ \lambda\_.e_2)$$

One can then verify for all $i = 1, 2$ *that* $D[E[\mathit{choice}[e_1, e_2]]$ *may reduce to* $(\nu u_i)(u_i{\Leftarrow}e_i \mid D[E[u_i]])$ *modulo garbage collection so the respective other branch cannot become active. As a consequence, the expression* $\mathit{choice}[\mathsf{unit}, \bot]$ *may terminate by choosing the first branch, or diverge by choosing the second one.*

$$
\begin{array}{llll}
\text{ECs} & E & ::= & x{\Leftarrow}\widetilde{E}\ , \qquad \widetilde{E} ::= [\,] \mid \widetilde{E}\,e \mid v\,\widetilde{E} \mid \mathbf{exch}(\widetilde{E}, e) \mid \mathbf{exch}(v, \widetilde{E}) \\
\text{Future ECs} & F & ::= & x{\Leftarrow}\widetilde{F}\ , \qquad \widetilde{F} ::= \widetilde{E}[[\,]\,v] \mid \widetilde{E}[\mathbf{exch}([\,], v)] \\
\text{Process ECs} & D & ::= & [\,] \mid p \mid D \mid D \mid p \mid (\nu x)D
\end{array}
$$

Fig. 3. Evaluation contexts

Reduction rules.

$$
\begin{array}{ll}
(\beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev})) & E[(\lambda y.e)\,v] \to (\nu y)(E[e] \mid y{\Leftarrow}v) \\
(\text{THREAD.NEW}(\mathsf{ev})) & E[\mathbf{thread}\,v] \to (\nu z)(E[z] \mid z{\Leftarrow}v\,z) \\
(\text{FUT.DEREF}(\mathsf{ev})) & F[x] \mid x{\Leftarrow}v \to F[v] \mid x{\Leftarrow}v \\
(\text{HANDLE.NEW}(\mathsf{ev})) & E[\mathbf{handle}\,v] \to (\nu z)(\nu z')(E[v\,z\,z'] \mid z'\,\mathsf{h}\,z) \\
(\text{HANDLE.BIND}(\mathsf{ev})) & E[x\,v] \mid x\,\mathsf{h}\,y \to E[\mathbf{unit}] \mid y{\Leftarrow}v \mid x\,\mathsf{h}\,\bullet \\
(\text{CELL.NEW}(\mathsf{ev})) & E[\mathbf{cell}\,v] \to (\nu z)(E[z] \mid z\,\mathsf{c}\,v) \\
(\text{CELL.EXCH}(\mathsf{ev})) & E[\mathbf{exch}(z, v_1)] \mid z\,\mathsf{c}\,v_2 \to E[v_2] \mid z\,\mathsf{c}\,v_1 \\
(\text{LAZY.NEW}(\mathsf{ev})) & E[\mathbf{lazy}\,v] \to (\nu z)(E[z] \mid z \overset{susp}{\Longleftarrow} v\,z) \\
(\text{LAZY.TRIGGER}(\mathsf{ev})) & F[x] \mid x \overset{susp}{\Longleftarrow} e \to F[x] \mid x{\Leftarrow}e
\end{array}
$$

Non-well-formed processes. We assume that the rules cannot be applied to non-well-formed processes.

Distinct variable convention. We assume that all processes to which rules apply satisfy the distinct variable convention, and that all new binders use fresh variables (z and z'). Reduction results will then also satisfy the distinct variable convention, except for FUT.DEREF(ev) where values with bound variables may be copied, and where α-renaming has to be performed before applying the next rule.

Closure. Rule application is closed under structural congruence and process ECs D. If $p_1 \equiv D[p_1']$, $p_1' \to p_2'$, and $D[p_2'] \equiv p_2$ then $p_1 \to p_2$.

Fig. 4. One-step reduction relation of $\lambda(\mathsf{fut})$ denoted by $\to$ or $\overset{\mathsf{ev}}{\longrightarrow}$

3 Observational Semantics

As usual, we want to consider two processes equivalent whenever it is impossible for an observer to distinguish them. We model the observer by contexts which test whether or not a process in that context terminates successfully. This raises two questions: firstly, which kinds of failure situations can arise in a concurrent calculus with futures, and secondly, how to treat computations that may diverge.

3.1 May-Must Program Equivalence

With regard to execution failure, we note that besides the possibilities of type mismatches (and the resulting stuck expressions) and deadlocks (arising from computations in concurrent threads blocking each other), one can also introduce a direct cyclic dependency between concurrent futures. In call-by-need lambda calculi this phenomenon is known as a *black hole*. We require that in a non-failing computation,

every future eventually refers to a "proper" value.

Formally, we call a process p *successful* (meaning it has terminated successfully), if and only if it is well-formed and for every component $x \Leftarrow e$ of p, the future x is bound (possibly via a chain $x \Leftarrow x_1 \mid x_1 \Leftarrow x_2 \mid \ldots \mid x_{n-1} \Leftarrow x_n$) to a non-variable value, a cell, a lazy future, a handle, or a handled future. For example, $x \Leftarrow \lambda y.y$ and $x \Leftarrow y \mid y \,\mathsf{c}\, z$ are successful, while $x \Leftarrow x$ (a black hole) and $x \Leftarrow yx \mid y \Leftarrow xy$ (a deadlocked process) are not successful.

Next, we address the question concerning the notion of observable termination that we adopt. Given a binary relation t, we write t^+, t^*, t^ϵ for the transitive, reflexive-transitive and reflexive closure of t, respectively.

Definition 3.1 *Let p be a process. We say that p is* may-convergent *($p\!\downarrow$) if there exists a sequence of reductions $p \to^* p'$ such that p' is successful. It is* must-convergent *($p\!\Downarrow$) if all reduction descendants p' of p are may-convergent. We say that p is* must-divergent *($p\!\Uparrow$) if it has no reduction descendant that succeeds. It is* may-divergent *($p\!\uparrow$) if some reduction descendant of p is must-divergent.*

Note that all processes p satisfy $p\!\Uparrow \Leftrightarrow \neg p\!\downarrow$ and $p\!\uparrow \Leftrightarrow \neg p\!\Downarrow$, and that non-well-formed processes are must-divergent.

Lemma 3.2 *If p contains a future cycle $x_1 \Leftarrow x_2 \mid x_2 \Leftarrow x_3 \mid \ldots \mid x_n \Leftarrow x_1$ then $p\!\Uparrow$. In particular, if $p\!\downarrow$ then p does not contain such a cycle.*

We write $\downarrow$ for the set of may-convergent processes, and $\Downarrow$ for the set of must-convergent processes. Let $P = \ \downarrow$ or $P = \ \Downarrow$. We define binary relations $\leq_P$ both for processes and expressions, such that for all $p, p' \in \mathit{Process}$ and $e, e' \in \mathit{Exp}$:

$$p \leq_P p' \quad \text{iff} \quad \forall D.\ D[p] \in P \Rightarrow D[p'] \in P$$

$$e \leq_P e' \quad \text{iff} \quad \forall C.\ C[e] \in P \Rightarrow C[e'] \in P$$

In particular, $e \leq_P e'$ iff $C[e] \leq_P C[e']$ for all C. The *contextual preorder* $\leq$ is the intersection of may- and must-contextual approximation $\leq_\downarrow$ and $\leq_\Downarrow$. *Contextual equivalence* $\sim$ is the equivalence relation $\leq \cap \geq$ induced by the contextual preorder $\leq$. It is easy to see that contextual equivalence $\sim$ on processes and expressions is a congruence, i.e., $\sim$ is an equivalence relation such that $e \sim e'$ implies $C[e] \sim C[e']$ for all contexts C; the same for processes. It is easy to verify that $\bot \not\sim p \sim p'$ implies that the introduced but not ν-bound variables are the same for p and p', the same for expressions. In this case $C[p]$ is well-formed iff $C[p']$ is well-formed.

3.2 Fairness

The reduction strategy of Alice ML is fair in that every redex will be reduced eventually. A corresponding notion of fairness for $\lambda(\mathsf{fut})$ can be imposed as a refinement of its reduction strategy, and translates into a property of reduction sequences.

Rules of $\lambda(\mathsf{fut})$ select one or two of the parallel components, of which at most one is rewritten while the other remains, perhaps modified, if the rules HANDLE.BIND(ev) or CELL.EXCH(ev) have been applied, since then the handle or the cell component are modified. A *redex* is a subexpression that can be rewritten by applying some reduction step, or a suspended thread that can be activated (by LAZY.TRIGGER).

Either a process is must-divergent, or every reduction step of $\lambda(\mathsf{fut})$ removes one

redex while preserving all others. It is not difficult to formalize this statement, but this would not yield new insights.

Definition 3.3 *A reduction sequence R starting from p is* fair *iff every redex is eventually reduced after a finite number of reduction steps in R. For a process p we define $p{\downarrow_{fair}}$ iff there is a fair reduction from p to a successful process, and $p{\Downarrow_{fair}}$ iff for every reduction $p \to^* p'$, we have $p'{\downarrow_{fair}}$.*

Example 3.4 *Let $K_{1!} = \lambda x.\lambda y.(x\,\mathbf{unit})$ and $K_2 = \lambda x\lambda y.y$. Let p the process $x_1{\Leftarrow}y\,y \mid z\,\mathsf{c}\,K_{1!} \mid x_2{\Leftarrow}\mathbf{exch}(z, K_2) \mid y{\Leftarrow}\lambda x.\mathbf{exch}(z, K_{1!})\,(\lambda\_.(x\,x))\,\mathbf{unit}$ which is must-convergent. There is a reduction to a successful process that first applies* CELL.EXCH(ev) *(so that the z-cell contains K_2), and subsequently reduces the x_1-thread to* **unit**. *In contrast, the unfair reduction never puts K_2 into the z-cell but always exchanges $K_{1!}$ with $K_{1!}$.*

Proposition 3.5 *Let p be a process. Then $p{\downarrow} \Leftrightarrow p{\downarrow_{fair}}$ and $p{\Downarrow} \Leftrightarrow p{\Downarrow_{fair}}$.*

Proof. The definition of fairness excludes certain infinite reductions and the notion of may- and must-convergence is founded on finite reductions. Thus, they are not affected by imposing fairness. □

Theorem 3.6 *We have $\leq_{\downarrow_{fair}} = \leq_{\downarrow}$ and $\leq_{\Downarrow_{fair}} = \leq_{\Downarrow}$, hence also $\sim$ is unchanged if the definition is restricted to fair reductions.*

3.3 Discussion

Neither may- nor must-convergence alone yields a satisfactory notion of observational equivalence, as the former cannot distinguish v from $choice[v, \bot]$ while the latter equates $choice[v, \bot]$ and $\bot$, where v is any value. Moreover, we believe that our must-convergence is conceptually more adequate than considering a total must-convergence, which enforces all reductions to successfully terminate.

We adapt the example of [2]: Let J be $\mathbf{thread}\ (\lambda f.\lambda x.choice[(f\ \ x), I])\,I$ and let I denote $\lambda x.x$. Since **thread** can be used for fixpointing, the process $u{\Leftarrow}J$ reduces to $u{\Leftarrow}z\,I \mid z{\Leftarrow}((\lambda f.\lambda x.choice[(f\ \ x), I])\ \ z)$. Our semantics yields

$$u{\Leftarrow}I \ \sim\ u{\Leftarrow}J \ \not\sim\ u{\Leftarrow}choice[\bot, I]$$

where the process $u{\Leftarrow}J$ is must-convergent, has an infinite reduction that is fair, and all successful results are equivalent to $u{\Leftarrow}I$. This means it is correct to introduce (or remove, respectively) weak divergences, but it is not correct to introduce (or remove, respectively) strong divergences (reduction possibilities to errors or must-divergent processes). However, if our must-convergence is replaced by total must-convergence, then this is reversed, and $u{\Leftarrow}I$ would be non-equivalent to $u{\Leftarrow}J$, which in turn would be equivalent to $u{\Leftarrow}choice[\bot, I]$.

There are also examples (see Example 3.4) of must-convergent processes with infinite reductions, where the infinite reduction are excluded if fairness is assumed.

We give a further argument for our definition of must-convergence. Suppose a process p can reduce to infinitely many different values $v_i, i \in \mathbb{N}$, but cannot reduce to an error, i.e. to a must-divergent process. Then, because of the finitely branching non-determinism of $\lambda(\mathsf{fut})$, p necessarily also permits an infinite reduction. Using our semantics, p is must-convergent. For any p' with $p' \sim p$ the process p' has the

possibilities to reduce to the same values as p, but not to an error. In contrast, replacing our must-convergence by total must-convergence in the definition of $\sim$, there may be $p' \sim p$ such that p' may reduce to an error, which is clearly undesirable.

3.4 Evaluation Contexts are Enough

The fundamental context lemma for expressions states that ECs provide already enough observations to distinguish observationally nonequivalent expressions and processes. Establishing equivalences is made considerably more tractable by the context lemma. It also supports proving equivalence of expressions as e.g. in Theorems 4.23 and 4.24 and in Subsection 4.5. We define contextual approximations in evaluation contexts for the set of processes $P = \ \downarrow$ and $P = \ \Downarrow$ as $e \leq_P^{\mathsf{ev}} e'$ iff $\forall E \forall D : D[E[e]] \in P \Rightarrow D[E[e']] \in P$.

Proposition 3.7 (Context Lemma) *For all $e_1, e_2 \in Exp$:*

$$e_1 \leq_{\downarrow}^{\mathsf{ev}} e_2 \quad \textit{and} \quad e_1 \leq_{\Downarrow}^{\mathsf{ev}} e_2 \quad \Rightarrow \quad e_1 \leq e_2$$

The proof can be found in Appendix A.

4 Program Transformations

We present a set of transformation rules that allow for partial evaluation, and show which of these reduction rules are correct. In particular we show that call-by-value beta reduction can be performed in arbitrary contexts.

Candidates of transformation rules are collected in Fig. 5. They are parametrized by strategies strat which fix the contexts in which the rule can be applied. We assume all transformations to be closed under structural congruence and process ECs. The strategy ev is the reduction strategy of $\lambda(\mathsf{fut})$. It permits ECs E for all rules but FUT.DEREF and LAZY.TRIGGER where it requires future ECs F. The strategy f permits all flat contexts, while d insists on deep contexts. Other strategies can be defined by Boolean combinations, for instance $\neg\mathsf{ev}\wedge\mathsf{f}$. In particular, the strategy with arbitrary contexts is $\mathsf{a} = \mathsf{f} \vee \mathsf{d}$.

The first set of transformation rules in Fig. 5 is obtained by lifting reduction rules of $\lambda(\mathsf{fut})$ from ECs to contexts permitted by the strategy. The second set contains call-by-value β-reduction in contexts permitted by the strategy, garbage collection, and deterministic cell exchange. The dereferencing of values into cells (CELL.DEREF) is included mainly for technical reasons.

Definition 4.1 *A transformation t is* correct *iff $(p, p') \in t$ implies that $p \sim p'$.*

First we state which transformations are *not* correct. In Appendix B we provide counter-examples for the transformations mentioned in the proposition below.

Proposition 4.2 *The transformations* CELL.EXCH(ev), LAZY.TRIGGER(f), HANDLE.NEW($\neg\mathsf{ev}$), HANDLE.BIND($\neg\mathsf{ev}$), CELL.NEW($\neg\mathsf{ev}$), THREAD.NEW($\neg\mathsf{ev}$), LAZY.NEW($\neg\mathsf{ev}$), *and the rule β-CBN with $C[(\lambda x.e)\ e'] \to C[e[e'/x]]$ are* not *correct.*

A helpful tool for proving correctness of transformations is the following lemma, which shows that it is not necessary to consider enclosing D-contexts if the transformation is alreay closed under all contexts D.

Lifting reduction to transformation rules.

$$
\begin{array}{ll}
(\beta\text{-CBV}_{\mathrm{L}}(\mathsf{strat})) & C[(\lambda y.e)\, v] \to (\nu y)(C[e] \mid y \Leftarrow v) \\
(\text{THREAD.NEW}(\mathsf{strat})) & C[\mathbf{thread}\, v] \to (\nu z)(C[z] \mid z \Leftarrow v\, z) \\
(\text{FUT.DEREF}(\mathsf{strat})) & C[x] \mid x \Leftarrow v \to C[v] \mid x \Leftarrow v \\
(\text{HANDLE.NEW}(\mathsf{strat})) & C[\mathbf{handle}\, v] \to (\nu z)(\nu z')(C[v\, z\, z'] \mid z'\, \mathsf{h}\, z) \\
(\text{HANDLE.BIND}(\mathsf{strat})) & C[x\, v] \mid y\, \mathsf{h}\, x \to C[\mathbf{unit}] \mid y \Leftarrow v \mid x\, \mathsf{h}\, \bullet \\
(\text{CELL.NEW}(\mathsf{strat})) & C[\mathbf{cell}\, v] \to (\nu z)(C[z] \mid z\, \mathsf{c}\, v) \\
(\text{CELL.EXCH}(\mathsf{strat})) & C[\mathbf{exch}(y, v_1)] \mid y\, \mathsf{c}\, v_2 \to C[v_2] \mid y\, \mathsf{c}\, v_1 \\
(\text{LAZY.NEW}(\mathsf{strat})) & C[\mathbf{lazy}\, v] \to (\nu z)(C[z] \mid z \overset{susp}{\Longleftarrow} v\, z) \\
(\text{LAZY.TRIGGER}(\mathsf{strat})) & C[x] \mid x \overset{susp}{\Longleftarrow} e \to C[x] \mid x \Leftarrow e
\end{array}
$$

Call-by-value beta reduction and other deterministic transformations.

$$
\begin{array}{ll}
(\beta\text{-CBV}(\mathsf{strat})) & C[(\lambda x.e)\ v] \ \to\ C[e[v/x]] \\
(\text{CELL.DEREF}) & p \mid y\, \mathsf{c}\, x \mid x \Leftarrow v \to\ p \mid y\, \mathsf{c}\, v \mid x \Leftarrow v \\
(\text{GC}) & p \mid (\nu y_1) \ldots (\nu y_n) p' \ \to\ p \qquad \text{if } p' \text{ is successful and} \\
 & y_1, \ldots, y_n \text{ contain all process variables of } p' \\
(\text{DET.EXCH}) & (\nu x)(y \Leftarrow \widetilde{E}[\mathbf{exch}(x, v_1)] \mid x\, \mathsf{c}\, v_2) \to\ (\nu x)(y \Leftarrow \widetilde{E}[v_2] \mid x\, \mathsf{c}\, v_1)
\end{array}
$$

No capturing. The same conditions as in Fig. 4 are assumed. In addition we assume that no variables are moved out of their scope or into the scope of some other binder, i.e., $\mathsf{fv}(v) \cap \mathsf{bv}(C) = \emptyset$, and that α-renaming is also done after CELL.DEREF.

Closure and Strategy. Transformations are always closed under structural congruence and D contexts. For all above rules $r(\mathsf{strat})$ the class of contexts C is restricted by the strategy strat. We write $p_1 \xrightarrow{r(\mathsf{strat})} p_2$ if $p_1 \to p_2$ by this rule.

Fig. 5. Transformation rules for some strategy strat permitting contexts C

Lemma 4.3 *A transformation t that is closed under all contexts D is correct iff it satisfies $p{\downarrow} \Leftrightarrow p'{\downarrow}$ and $p{\uparrow} \Leftrightarrow p'{\uparrow}$ for all pairs $(p, p') \in t$.*

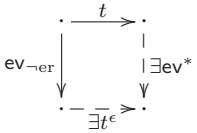
In the remaining subsections we will prove the correctness of various program transformations. In Subsection 4.1 we prove a lemma which implies the correctness of all deterministic reductions. In Subsection 4.2 we provide a sufficient criterion for correctness of a program transformation (Lemma 4.7). We will show that GC and DET.EXCH obviously satisfy this criterion and hence are correct. In Subsection 4.3 we extend our proof technique with the notions of complete sets of forking and commuting diagrams which will enable us to proof the correctness of β-CBV$_{\mathrm{L}}(\mathsf{f})$, FUT.DEREF($\mathsf{f}$) and CELL.DEREF. Unfortunately, the transformations β-CBV$_{\mathrm{L}}(\mathsf{d})$ and FUT.DEREF(d) do not meet the conditions of Lemma 4.7. Hence, in Subsection 4.4 we give a fully worked out proof of correctness using a refined induction proof method. After presenting some consequences of the correctness of β-CBV$_{\mathrm{L}}(\mathsf{a})$ we finish this section with an exemplary application (unnecessary cell allocation) of our results. In all of our proofs we use two sets of reduction sequences:

Definition 4.4 *Let p be a process. With* $\mathrm{Suc}(p)$ *(*$\mathrm{Div}(p)$*, respectively) we denote all sequences of reductions for p that end in a successful process (must-divergent process, respectively).*

4.1 Correctness of Deterministic Reductions

We show that all deterministic reduction steps of $\lambda(\mathsf{fut})$ are correct transformations. This excludes the rule CELL.EXCH(ev), the only source of non-determinism in $\lambda(\mathsf{fut})$. The proof relies on the diagrams used in [12] to show the uniform confluence of the fragment of $\lambda(\mathsf{fut})$ without cell exchange and handle-errors.
Let ev be the reduction $\xrightarrow{\mathsf{ev}}$ of $\lambda(\mathsf{fut})$ and $p \xrightarrow{\mathsf{ev}_{\neg\mathrm{er}}} p'$ iff $p \xrightarrow{\mathsf{ev}} p'$ and p is may-convergent.

Lemma 4.5 *A transformation t on processes that is closed under D-contexts and subsumed by reduction $t \subseteq \mathsf{ev}$ is correct if it satisfies the forking condition* $\mathsf{ev}_{\neg er}^{-1} \circ t \subseteq t^{\epsilon} \circ (\mathsf{ev}^*)^{-1}$.



Proof. Since t is closed under D-contexts it suffices to show for all $(p, p') \in t$ that p and p' have identical may- and must-convergence behaviour. That $p'{\downarrow} \Rightarrow p{\downarrow}$ and $p'{\uparrow} \Rightarrow p{\uparrow}$ is obvious since $t \subseteq \mathsf{ev}$. We prove the remaining cases:

$\mathbf{p}{\downarrow} \Rightarrow \mathbf{p'}{\downarrow}$: Since $p{\downarrow}$, there exists $R \in \mathrm{Suc}(p)$. The proof is by induction on the length of R, which cannot be 0 since $t \subseteq \mathsf{ev}$. For the inductive step, we use the forking condition on the first reduction of R. If the diagram is closed by an $\xleftarrow{\mathsf{ev}^*}$ step then $p'{\downarrow}$, otherwise the induction hypothesis applies.

$\mathbf{p}{\uparrow} \Rightarrow \mathbf{p'}{\uparrow}$: By induction on the length of a minimal reduction sequence $R \in \mathrm{Div}(p)$. If the length is 0, then p is must-divergent and so $p'{\Uparrow}$, since we have already established $p'{\downarrow} \Rightarrow p{\downarrow}$. Otherwise, p is may-convergent, so that we can apply the forking diagram. The rest follows from the induction hypothesis. □

Note that t preserves must-divergence since reduction ev does. If t raises a handle error, i.e. generates components of the form $E[z\ v_1] \mid z\,\mathsf{h}\,\bullet$, then the result is a must-divergent process.

Proposition 4.6 *All reduction steps of* $\lambda(\mathsf{fut})$ *are correct program transformations except for* CELL.EXCH*(*ev*).*

Proof. The diagrams required by Lemma 4.5 can be shown as in [12], with a slight modification: Instead of call-by-value beta reduction one needs to consider β-$\mathrm{CBV_L}(\mathsf{ev})$ and additionally the overlappings with rule CELL.NEW(ev). Both modifications are easy to handle. The only rule for which some care is needed is the rule HANDLE.BIND(ev). This rule can introduce non-determinism, but only when raising handle errors which results in a must-divergent process: a typical counter example is $E_1[z\ v_1] \mid E_2[z\ v_2] \mid z\,\mathsf{h}\,y$ which has two reducts $E_1[\mathbf{unit}] \mid y{\Leftarrow}v_1 \mid E_2[z\ v_2] \mid z\,\mathsf{h}\,\bullet$ and $E_1[z\ v_1] \mid y{\Leftarrow}v_2 \mid E_2[\mathbf{unit}] \mid z\,\mathsf{h}\,\bullet$ that cannot be joined, but both constitute so-called handle-errors, which cannnot be reduced to successful processes. The rule commutes with itself in case no handle error is raised. □

4.2 A Sufficient Criterion for Correctness

The following lemma provides three conditions of a transformation which ensure correctness of the transformation. We will use the lemma in this and the subsequent subsection to show that GC, DET.EXCH, β-CBV$_{\mathrm{L}}(\mathsf{f})$, FUT.DEREF($\mathsf{f}$) and CELL.DEREF are correct program transformations.

Lemma 4.7 *A transformation t on processes is correct if it satisfies the following three conditions:*

(**fork**) $\mathsf{ev}^{-1} \circ t \;\subseteq\; t^* \circ (\mathsf{ev}^{\epsilon})^{-1}$ (**commute**) $t \circ \mathsf{ev} \;\subseteq\; \mathsf{ev}^* \circ t^{\epsilon}$
(**success**) *for all $(p, p') \in t$: p is successful iff p' is successful and $(p, p') \notin \mathsf{ev}$.*

Proof. Since t is closed under D-contexts, it is sufficient to show for all p, p' with $(p, p') \in t$ that p and p' have identical may- and must-convergence behaviour.

$\mathbf{p}\!\downarrow \Rightarrow \mathbf{p}'\!\downarrow$: By induction on the length of $R \in \mathrm{Suc}(p)$, we show that there exists $R' \in \mathrm{Suc}(p')$ of smaller or equal length. In the base case of length 0, p is successful and thus p' by condition (**success**). Otherwise consider the first reduction step. There exists p_1 such that $p \xrightarrow{\mathsf{ev}} p_1$ and p_1 has a smaller successful reduction sequence. Thus, we can apply condition (**fork**) for some p_1' with $p' \xrightarrow{\mathsf{ev}^{\epsilon}} p_1'$ and $p_1 \xrightarrow{t^*} p_1'$. We conclude the proof by induction on the length of $p_1 \xrightarrow{t^*} p_1'$. If this length is 0 then $p_1 \equiv p_1'$ so p' has a successful reduction sequence of length smaller or equal to that of p. If the length is n, we apply the first induction hypothesis to the first transformation step, and use the other induction hypothesis for the remaining sequence of $n-1$ transformation steps.

$\mathbf{p}'\!\downarrow \Rightarrow \mathbf{p}\!\downarrow$: By induction on the length of $R \in \mathrm{Suc}(p')$. The case $p \xrightarrow{\mathsf{ev}} p'$ is obvious, so we can assume $(p, p') \notin \mathsf{ev}$. In the base case, this length is 0 so p' is successful. Assumption (**success**) implies that p is successful too. For larger lengths, we can apply the (**commute**) condition, and then the induction hypothesis.

$\mathbf{p}\!\uparrow \Rightarrow \mathbf{p}'\!\uparrow$: By induction on the length of $R \in \mathrm{Div}(p)$ we show that there exists $R' \in \mathrm{Div}(p')$ of smaller or equal length. In the base case, $p\Uparrow$, hence $p'\Uparrow$ as shown in case $\mathbf{p}'\!\downarrow \Rightarrow \mathbf{p}\!\downarrow$. The induction step uses the (**fork**) diagram.

$\mathbf{p}'\!\uparrow \Rightarrow \mathbf{p}\!\uparrow$: By induction on the length of $R \in \mathrm{Div}(p')$. In the base case, p' must-diverges and so does p as we showed in case $\mathbf{p}\!\downarrow \Rightarrow \mathbf{p}'\!\downarrow$. The induction step relies on the (**commute**) diagram. □

Now it is easy to show that garbage collection (GC) and deterministic cell exchange (DET.EXCH) are correct, since the overlappings of these transformations with reductions are trivial.

Theorem 4.8 GC *and* DET.EXCH *are correct program transformations.*

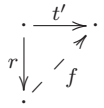
Proof. This follows by Lemma 4.7, since GC has no influence on reduction sequences, i.e. $\mathsf{ev}^{-1} \circ \mathrm{GC} \subseteq \mathrm{GC} \circ \mathsf{ev}^{-1}$ and $\mathrm{GC} \circ \mathsf{ev} \subseteq \mathsf{ev} \circ \mathrm{GC}$ and since the conditions ensure that there is no interference with the success of processes. In the same way

the correctness of DET.EXCH follows. □

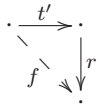
4.3 Correctness of β-CBV$_{\mathrm{L}}$(f), FUT.DEREF(f) and CELL.DEREF

For further proofs we require the notion of sets of forking and commuting diagrams, which is a formalism to represent the overlappings between a transformation rule and reductions. Informally, the completeness condition ensures that every non-trivial overlapping is covered by the set of diagrams.

Definition 4.9 Forking *and* commuting diagrams *for a transformation t are meta-rewriting rules for some $r \subseteq \mathsf{ev}$, $t' \subseteq t$ and f being relations on processes.*



forking diagram



commuting diagram

A set of forking diagrams $\{\xleftarrow{r_1} \cdot \xrightarrow{t_1} \rightsquigarrow \xrightarrow{f_1}, \ldots, \xleftarrow{r_n} \cdot \xrightarrow{t_n} \rightsquigarrow \xrightarrow{f_n}\}$ *is* complete *iff for every reduction sequence* $p_1 \xleftarrow{\mathsf{ev}} p_2 \xrightarrow{t} p_3$ *there exists a forking diagram* $\xleftarrow{r_i} \cdot \xrightarrow{t_i} \rightsquigarrow \xrightarrow{f_i}$ *with* $(p_1, p_2) \in r_i^{-1}$, $(p_2, p_3) \in t_i$ *and* $(p_1, p_3) \in f_i$.

A set of commuting diagrams $\{\xrightarrow{t_1} \cdot \xrightarrow{r_1} \rightsquigarrow \xrightarrow{f_1}, \ldots, \xrightarrow{t_n} \cdot \xrightarrow{r_n} \rightsquigarrow \xrightarrow{f_n}\}$ *is* complete *iff for every reduction sequence* $p_1 \xrightarrow{t} p_2 \xrightarrow{\mathsf{ev}} p_3$ *there exists a commuting diagram* $\xrightarrow{t_i} \cdot \xrightarrow{r_i} \rightsquigarrow \xrightarrow{f_i}$ *with* $(p_1, p_2) \in t_i, (p_2, p_3) \in r_i$ *and* $(p_1, p_3) \in f_i$.

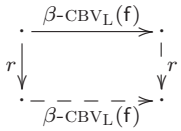
In the remaining part of this subsection we compute complete sets of commuting and forking diagrams for the transformations β-CBV$_{\mathrm{L}}$(f), CELL.DEREF and FUT.DEREF(f) and show that the diagrams meet the conditions (**fork**) and (**commute**) of Lemma 4.7. The third condition (**success**) is proved by an additional lemma for each transformation.

First we treat the transformation β-CBV$_{\mathrm{L}}$(f), i.e. call-by-value β-reduction not inside the body of an abstraction. Correctness of arbitrary call-by-value β-reduction (particularly inside deep contexts) will be proved in subsection 4.4. The following lemma shows that condition (**success**) of Lemma 4.7 holds for β-CBV$_{\mathrm{L}}$(f).

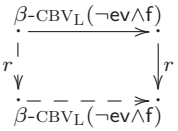
Lemma 4.10 *Let p_1, p_2 be two configurations with $p_1 \xrightarrow{\beta\text{-CBV}_{\mathrm{L}}(\neg\mathsf{ev})} p_2$. Then p_1 is successful iff p_2 is successful.*

A case analysis of the overlappings between reductions and the transformation β-CBV$_{\mathrm{L}}$(f) shows the following lemma:

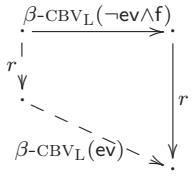
Lemma 4.11 *A complete set of forking diagrams for β-CBV$_{\mathrm{L}}$(f) and a complete set of commuting diagrams for β-CBV$_{\mathrm{L}}$($\neg$ev$\wedge$f) are:*



for every reduction r



for every reduction r



for $r \in$
$\{\beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$

Proposition 4.12 *β-$\mathrm{CBV}_{\mathrm{L}}(\mathsf{f})$ is a correct transformation.*

Proof. From Lemma 4.7 which is applicable by Lemmas 4.10 and 4.11 and from the fact that $\beta\text{-}\mathrm{CBV}_{\mathrm{L}}(\mathsf{ev}) \circ \mathsf{ev} \subseteq \mathsf{ev}^*$. □

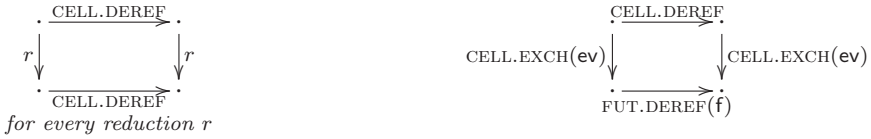
The proof of correctness of the transformation FUT.DEREF(f), i.e. copying of values into flat contexts requires the transformation CELL.DEREF, since it may happen that an overlapping of a FUT.DEREF(f)-transformation with a reduction can only be closed using copying of values into a cell. Hence, we will prove the union of both transformations being correct.

Just as before we first prove the condition (**success**) required by Lemma 4.7.

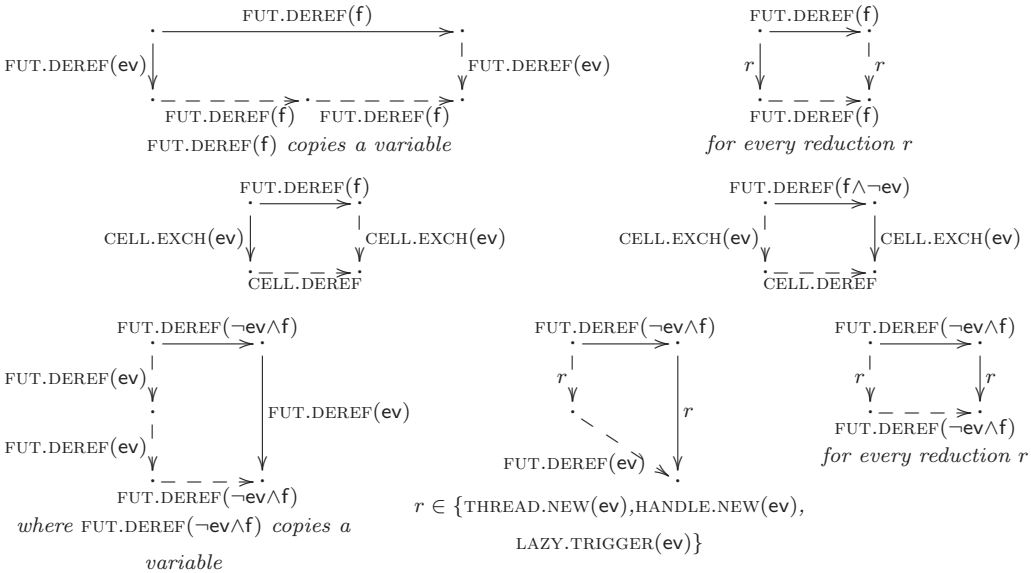
Lemma 4.13 *Let p_1, p_2 be processes with $p_1 \xrightarrow{\text{FUT.DEREF}(\neg\mathsf{ev})} p_2$ or $p_1 \xrightarrow{\text{CELL.DEREF}} p_2$, then p_1 is successful iff p_2 is successful.*

The next two lemmas shows the forking and commuting diagrams for both transformations. Completeness follows by a case analysis of the overlappings between a transformation and a reduction. In Appendix C exemplary cases are shown.

Lemma 4.14 *The forking and commuting diagrams for* CELL.DEREF *can be read off the following diagrams:*



Lemma 4.15 *A complete set of forking diagrams for* FUT.DEREF(f) *and a complete set of commuting diagrams for* FUT.DEREF($\neg$ev$\wedge$f) *is given by the following diagrams:*



From Lemma 4.7 which is applicable by combining the diagrams of Lemmas 4.14, 4.13 and 4.15 and the fact that FUT.DEREF(ev) $\circ$ ev $\subseteq$ ev* the following proposition holds:

Proposition 4.16 FUT.DEREF(f) *and* CELL.DEREF *are correct transformations.*

4.4 Correctness of FUT.DEREF(a) *and* β-CBV(a)

We strengthen this result for FUT.DEREF(f) in order to prove the correctness of FUT.DEREF(d) wrt. suitable measures on terms and sequences of reductions.

Definition 4.17 *Let p be a process with $p \equiv x_1 \Leftarrow x_2 \mid x_2 \Leftarrow x_3 \mid \ldots x_n \Leftarrow e \mid p'$ where e is not a variable and the chain of variables is maximal. Then the measure* $\mathrm{cl} : \mathit{Var} \to \mathbb{N}_0$ *is defined as* $\mathrm{cl}(x_1) := n$. *If the chain contains a chain of cyclic thread components $x_i \Leftarrow x_{i+1} \mid \ldots \mid x_j \Leftarrow x_i$ then $\mathrm{cl}(x_1)$ is undefined. The measure* $\#\mathrm{var}_\mathsf{f} : \mathit{Process} \to \mathbb{N}_0$ *is defined as follows: Let p be a process, then $\#\mathrm{var}_\mathsf{f}(p)$ is the sum of $\mathrm{cl}(x)$ of all occurrences of variables x in p, where the occurrence of x is inside a flat context.*

Let $R = p_0 \xrightarrow{\mathsf{ev}} p_1 \ldots \xrightarrow{\mathsf{ev}} p_n$ be a sequence of reductions. Then $\mathrm{rl}(R)$ is the number of reductions of R, i.e. $\mathrm{rl}(R) = n$, and $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R)$ is the number of reductions r of R with $r \not\subseteq$ FUT.DEREF(ev).

Lemma 4.18 *Let p, p' be two processes with $p \xrightarrow{\text{FUT.DEREF}(\mathsf{f})} p'$.*

i) If there exists $R \in \mathrm{Suc}(p)$ ($R \in \mathrm{Div}(p)$, resp.) then there exists $R' \in \mathrm{Suc}(p')$ ($R' \in \mathrm{Div}(p)$, resp.) with $\mathrm{rl}(R') \leq \mathrm{rl}(R)$.

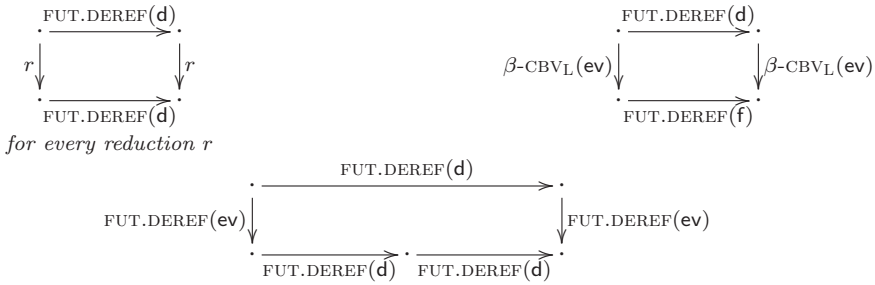
ii) If there exists $R' \in \mathrm{Suc}(p')$ ($R' \in \mathrm{Div}(p')$, resp.) then there exists $R \in \mathrm{Suc}(p)$ ($R \in \mathrm{Div}(p)$, resp.) with $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R) \leq \mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(R')$.

Proof. Follows by inspecting the forking and commuting diagrams for FUT.DEREF(f) that are used for the construction of the reduction sequences. □

Lemma 4.19 *If p is a process without cyclic chains of threads, then every* FUT.DEREF(d) *transformation preserves the measure* $\#\mathrm{var}_\mathsf{f}$.

A case analysis of the overlappings between the transformation FUT.DEREF(d) and reductions shows the following lemma. Examples for the cases covered by the diagrams can be found in Appendix C.

Lemma 4.20 *Complete sets of forking and commuting diagrams for* FUT.DEREF(d) *can be read off the following diagrams for all reductions r:*



Note that the last diagram read as commuting diagram breaks the condition (**commute**) of Lemma 4.7. Hence, the lemma is not applicable. We prove the correctness by induction on a combination of the measures $\#\mathrm{var}_\mathsf{f}, \mathrm{rl}(\cdot)$ and $\mathrm{rl}_{\neg \text{FUT.DEREF}(\mathsf{ev})}(\cdot)$ from Definition 4.17.

Proposition 4.21 *The transformation* FUT.DEREF(d) *is correct.*

Proof. Let $p_1 \xrightarrow{\text{FUT.DEREF(d)}} p_2$. We split the proof into four parts:

$\mathbf{p_1}\downarrow \Rightarrow \mathbf{p_2}\downarrow$: Let $R \in \text{Suc}(p_1)$. We show by induction on $l = \text{rl}(R)$ that there also exists $R' \in \text{Suc}(p_2)$ with length $\leq l$.

Lemma 4.13 implies that the base case holds. Now let $l > 0$. Then we can apply one of the forking diagrams of Lemma 4.20 to a suffix of the sequence $\xleftarrow{R} p_1 \xrightarrow{\text{FUT.DEREF(d)}} p_2$ and then use the induction hypothesis. For the second diagram of Lemma 4.20 we apply Lemma 4.18, and for the last diagram we apply the induction hypothesis twice.

$\mathbf{p_2}\downarrow \Rightarrow \mathbf{p_1}\downarrow$: We use the (lexicographically ordered) measure (μ_1, μ_2) on reduction sequences of the form $p_1 \xrightarrow{\text{FUT.DEREF(a)}} p_2 \xrightarrow{R}$ with $R \in \text{Suc}(p_2)$, $\mu_1 = \text{rl}_{\neg\text{FUT.DEREF(ev)}}(R)$, and $\mu_2 = \#\text{var}_\mathsf{f}(p_2)$. Note that μ_2 is defined, since by Lemma 3.2 the corresponding process does not contain a cyclic chain of threads.

Let $R \in \text{Suc}(p_2)$. We show by induction on (μ_1, μ_2) that there exists $R' \in \text{Suc}(p_1)$ with $\text{rl}_{\neg\text{FUT.DEREF(ev)}}(R') \leq \text{rl}_{\neg\text{FUT.DEREF(ev)}}(R)$.

For the base case let $(\mu_1, \mu_2) = (0, 0)$. Then Lemma 4.19 implies that R must be empty. Hence, p_2 is a successful process and Lemma 4.13 shows the claim. For the induction step let $(\mu_1, \mu_2) > (0, 0)$. We apply a commuting diagram from Lemma 4.20 to the sequence $p_1 \xrightarrow{\text{FUT.DEREF(d)}} p_2 \xrightarrow{R}$.

- If the first diagram is applicable, and the first reduction of R is a FUT.DEREF(ev) then μ_1 is unchanged, but μ_2 is strictly decreased. Otherwise μ_1 is strictly decreased. Hence we can apply the induction hypothesis.
- In case of the second diagram, μ_1 is strictly decreased and Lemma 4.18 shows the existence of R' with $\text{rl}_{\neg\text{FUT.DEREF(ev)}}(R') \leq \text{rl}_{\neg\text{FUT.DEREF(ev)}}(R)$.
- In case of the last diagram we can apply the induction hypothesis twice, since FUT.DEREF(ev) decreases the measure μ_2 and leaves μ_1 unchanged and a FUT.DEREF(d) transformation does not change μ_2 (see Lemma 4.19).

In any case, the constructed reduction sequence satisfies $\text{rl}_{\neg\text{FUT.DEREF(ev)}}(R') \leq \text{rl}_{\neg\text{FUT.DEREF(ev)}}(R)$.

$\mathbf{p_1}\uparrow \Rightarrow \mathbf{p_2}\uparrow$: This follows by induction on the length of a sequence $R \in \text{Div}(p_1)$ and by using the forking diagrams. The base case follows from the previous case, $p_2\downarrow \Rightarrow p_1\downarrow$. The induction step is analogous to the first case of the proof.

$\mathbf{p_2}\uparrow \Rightarrow \mathbf{p_1}\uparrow$: This follows by induction on the measure (μ_1, μ_2) where $\mu_1 = \text{rl}_{\neg\text{FUT.DEREF(ev)}}(R)$ with $R \in \text{Div}(p_2)$ being a shortest sequence of reductions and $\mu_2 = \#\text{var}_\mathsf{f}(p_2)$. Note that μ_2 may be undefined, but only for the last contractum of R, since R is a shortest sequence. Moreover, it is necessary to observe that FUT.DEREF(d) does not introduce cyclic chains of threads. The base case, i.e. $p_2\Uparrow$ follows from the first case of the proof, $p_1\downarrow \Rightarrow p_2\downarrow$. The induction step is analogous to the second case, using the commuting diagrams. □

Since FUT.DEREF(d) and FUT.DEREF(f) are correct program transformations, a summarizing theorem is:

Theorem 4.22 FUT.DEREF(a) *is a correct program transformation.*

Now we lift the result of correctness of call-by-value β-reduction inside flat contexts to arbitrary contexts, using the context lemma and the correctness of FUT.DEREF(a).

Theorem 4.23 *β-CBV(a) is a correct program transformation.*

Proof. By the context lemma it suffices to show that β-CBV(f) is correct. In all flat contexts C, the transformation $C[(\lambda x.e)\ v] \xrightarrow{\beta\text{-CBV(f)}} C[e[v/x]]$ can be replaced by the sequence of transformations

$$\begin{array}{ll} C[(\lambda x.e)\, v] & \xrightarrow{\beta\text{-CBV}_{\mathrm{L}}(\mathsf{f})} \ (\nu x)\ C[e] \mid x{\Leftarrow}v \\ & \xrightarrow{\text{FUT.DEREF(a)}}{}^{*}\ (\nu x)\ C[e[v/x]] \mid x{\Leftarrow}v \xrightarrow{\text{GC}} C[e[v/x]] \end{array}$$

Since we have shown that β-CBV$_{\mathrm{L}}$(f), FUT.DEREF(a) and GC are correct in Theorem 4.8 and Proposition 4.12, respectively, the result follows. □

Another consequence of the correctness of FUT.DEREF(a) and GC is:

Theorem 4.24 *Path compression, $(\nu y)(x{\Leftarrow}y \mid y{\Leftarrow}v) \to x{\Leftarrow}v$ where $y \notin \mathsf{fv}(v)$, is correct.*

The theorems also imply that the following equivalence holds, which is not covered by the congruence property, since $x\,\mathsf{c}\,[]$ is not a context: $v_1 \sim v_2 \implies p \mid x\,\mathsf{c}\,v_1 \sim p \mid x\,\mathsf{c}\,v_2$, which follows from the equivalence $p \mid ((\nu y)(x\,\mathsf{c}\,y \mid y{\Leftarrow}v)) \sim p \mid x\,\mathsf{c}\,v$.

4.5 An Example: Avoiding Unnecessary Cell Allocation

As an example application of our results we show that the function $\lambda x.\lambda y.(\mathit{let}\ z = (\mathbf{cell}\ x)\ \mathit{in}\ y\,(\mathbf{exch}(z, \mathbf{unit}))$ can be optimized to $\lambda x.\lambda y.(y\ x)$, by removing unnecessary cell allocations that are justified as correct transformation.

For all values v and expressions e we show $\mathit{let}\ z = (\mathbf{cell}\ v)\ \mathit{in}\ e\,(\mathbf{exch}(z, \mathbf{unit})) \sim (e\ v)$: Note that the let-expression on the left hand side is defined as being equivalent to $(\lambda z.e\,(\mathbf{exch}(z, \mathbf{unit})))\ (\mathbf{cell}\ v)$. Now let D be an arbitrary process evaluation context and E be an arbitrary evaluation context. We transform the process $D[E[(\lambda z.e\,(\mathbf{exch}(z, \mathbf{unit})))\ (\mathbf{cell}\ v)]]$ as follows:

$$\begin{array}{ll} & \xrightarrow{\text{CELL.NEW(ev)}}\ D[\nu x(E[(\lambda z.e\,(\mathbf{exch}(z, \mathbf{unit})))\ x] \mid x\,\mathsf{c}\,v)] \\ & \xrightarrow{\beta\text{-CBV(a)}}\ D[\nu x(E[(e\,(\mathbf{exch}(x, \mathbf{unit})))] \mid x\,\mathsf{c}\,v)] \\ & \xrightarrow{\text{DET.EXCH}}\ D[\nu x(E[(e\,v))] \mid x\,\mathsf{c}\,\mathbf{unit})] \xrightarrow{\text{GC}} D[E[(e\,v)]] \end{array}$$

Since we only used correct program transformation we have $D[E[\mathit{let}\ z = (\mathbf{cell}\ v)\ \mathit{in}\ e\,(\mathbf{exch}(z, \mathbf{unit}))] \sim D[E[(e\ v)]]$. An immediate consequence is $\mathit{let}\ z = (\mathbf{cell}\ v)\ \mathit{in}\ e\,(\mathbf{exch}(z, \mathbf{unit}))\ \ \eta\ \ (e\ v)$ for $\eta \in \{\leq^{\mathsf{ev}}_{\downarrow}, \leq^{\mathsf{ev}}_{\Downarrow}, (\leq^{\mathsf{ev}}_{\downarrow})^{-1}, (\leq^{\mathsf{ev}}_{\Downarrow})^{-1}\}$. Finally we apply the context lemma and have $\mathit{let}\ z = (\mathbf{cell}\ v)\ \mathit{in}\ e\,(\mathbf{exch}(z, \mathbf{unit})) \sim (e\ v)$. This finally proves that the optimization is correct, since $\sim$ is closed under arbitrary contexts.

Conclusions and Outlook

We have presented an observational equivalence for $\lambda(\mathsf{fut})$ programs, which allows us to reason about the correctness of transformations of stateful and concurrent computations, as found in the Alice ML core language [16,12]. Specifically, we have proved correctness of partial evaluation with respect to this semantics. In future work, we plan to investigate static analyses for $\lambda(\mathsf{fut})$, e.g. an adaptation of the calculus where touch optimization can be investigated [6]. Applying the correctness criterion of must- and may-convergence to optimizations of the reduction strategy also deserves attention.

References

[1] Baker, H. and C. Hewitt, *The incremental garbage collection of processes*, ACM Sigplan Notices **12** (1977), pp. 55–59.

[2] Carayol, A., D. Hirschkoff and D. Sangiorgi, *On the representation of McCarthy's amb in the pi-calculus.*, Theoret. Comput. Sci. **330** (2005), pp. 439–473.

[3] Chailloux, E., P. Manoury and B. Pagano, "Developing Applications With Objective Caml," O'Reilly, 2000, available online at http://caml.inria.fr/oreilly-book.

[4] Conchon, S. and F. L. Fessant, *Jocaml: Mobile agents for Objective-Caml*, in: *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, 1999.

[5] Ferreira, W., M. Hennessy and A. Jeffrey, *A theory of weak bisimulation for Core CML*, J. Funct. Programming **8** (1998), pp. 447–491.

[6] Flanagan, C. and M. Felleisen, *The semantics of future and an application*, J. Funct. Programming **9** (1999), pp. 1–31.

[7] Jeffrey, A. and J. Rathke, *A theory of bisimulation for a fragment of concurrent ML with local names*, Theoret. Comput. Sci. **323** (2004), pp. 1–48.

[8] Kutzner, A. and M. Schmidt-Schauß, *A nondeterministic call-by-need lambda calculus*, in: *International Conference on Functional Programming 1998* (1998), pp. 324–335.

[9] Milner, R., "Communicating and Mobile Systems: the π-Calculus," Cambridge University Press, 1999.

[10] Milner, R., M. Tofte, R. Harper and D. B. MacQueen, "The Standard ML Programming Language (Revised)," MIT Press, 1997.

[11] Moran, A. K., "Call-by-name, Call-by-need, and McCarthy's Amb," Ph.D. thesis, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden (1998).

[12] Niehren, J., J. Schwinghammer and G. Smolka, *A concurrent lambda calculus with futures*, Theoret. Comput. Sci.**364** (2006), pp. 338–356.

[13] Pitts, A. M., *Operational semantics and program equivalence*, in: J. T. O'Donnell, editor, *Applied Semantics*, LNCS **2395** (2002), pp. 378–412.

[14] Plotkin, G. D., *A powerdomain construction*, SIAM J. Comput. **5** (1976), pp. 452–487.

[15] Rossberg, A., *The missing link: dynamic components for ML*, in: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (2006), pp. 99–110.

[16] Rossberg, A., D. L. Botlan, G. Tack, T. Brunklaus and G. Smolka, "Alice Through the Looking Glass," Trends in Functional Programming **5**, Intellect Books, Bristol, UK, Munich, Germany, 2006 pp. 79–96.

[17] Sabel, D. and M. Schmidt-Schauß, *A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations*, Math. Structures Comput. Sci. (2007), accepted.

[18] Sangiorgi, D., *The lazy lambda calculus in a concurrency scenario*, Inform. and Comput. **111** (1994), pp. 120–153.

[19] Sangiorgi, D. and D. Walker, "The π-calculus: a Theory of Mobile Processes," Cambridge University Press, 2001.

[20] Smolka, G., *The Oz programming model*, in: J. van Leeuwen, editor, *Computer Science Today*, LNCS **1000**, Springer, 1995 pp. 324–343.

A Context Lemma for Expressions

The context lemma for expressions says that ECs provide enough observations to distinguish nonequivalent expressions. It talks about the relations

$$e \leq^{\mathsf{ev}}_{\downarrow} e' \text{ iff } \forall E \forall D : D[E[e]]\downarrow \Rightarrow D[E[e']]\downarrow$$
$$e \leq^{\mathsf{ev}}_{\Downarrow} e' \text{ iff } \forall E \forall D : D[E[e]]\Downarrow \Rightarrow D[E[e']]\Downarrow$$

Proposition A.1 (Context Lemma for $\leq_{\downarrow}$) *For all expressions e_1, e_2:*

$$e_1 \leq^{\mathsf{ev}}_{\downarrow} e_2 \;\Rightarrow\; e_1 \leq_{\downarrow} e_2$$

This will follow from Lemma A.3. In a first step, we have to generalize the context lemma for expressions to multicontexts, which may have more than one hole, or none at all. A *multicontext* M with n holes is a process that permits additional constants $[]_1, \ldots, []_n$ for marking holes in expression positions, each of which occurs exactly once. Similar as for contexts C, the component $x \,\mathsf{c}\, []$ is not possible. We write $M[e_1, \ldots, e_n]$ for the process obtained by replacing $[]_i$ by e_i for all $1 \leq i \leq n$. Note that e.g. the terms $D[C_1[[]_1] \mid \ldots \mid C_n[[]_n]]$ are multicontexts with n holes. For instance, if M is $z \,\mathsf{c}\, \lambda x.([]_1\ []_2) \mid y \Leftarrow []_3$ then $M[x, y, z]$ becomes $z \,\mathsf{c}\, \lambda x.x\ y \mid y \Leftarrow z$ We assume $[] = []_1$ so that standard contexts C become multicontexts. The i-th hole of a multicontext M is *in EC position* if $M[e_1, \ldots, e_{i-1}, [], e_{i+1}, \ldots, e_n]$ is a (process) EC of the form $D[E]$ for some expressions $e_1, \ldots, e_n$.

Lemma A.2 *If the i-th hole of M is in EC position then there exists an index j such that $M[e_1, \ldots, e_{j-1}, [\,], e_{j+1}, \ldots, e_n]$ is an EC for all expressions $e_1, \ldots, e_n$.*

Proof. We prove the corresponding property for *expression* multicontexts, by an induction on such contexts $\widetilde{M}$. In the case where $n \leq 1$ the proposition clearly holds. So suppose $\widetilde{M}$ has at least two holes. Then either $\widetilde{M}$ is of the form $\lambda x.\widetilde{M_1}$ or $\widetilde{M_1}\,\widetilde{M_2}$ or $\mathbf{exch}(\widetilde{M_1}, \widetilde{M_2})$. We may assume without loss of generality that for some $1 \leq k \leq n$, $\widetilde{M_1}$ is a multicontext over holes $[]_1, \ldots, []_k$, and $\widetilde{M_2}$ is a multicontext over $[]_{k+1}, \ldots, []_n$; otherwise we rename the holes accordingly.

- The case $\lambda x.\widetilde{M_1}$ is not possible, since every instantiation of $n-1$ holes of M_1 yields a deep context. In particular, this cannot be an EC.
- In the case $\widetilde{M_1}\,\widetilde{M_2}$, we distinguish two subcases: First, if there exist $e_1, \ldots, e_k$ and $1 \leq i \leq k$ such that $\widetilde{M_1}[e_1, \ldots, e_{i-1}, [\,], e_{i+1}, \ldots, e_k]$ is an EC, then by induction hypothesis there exists $1 \leq j \leq k$ such that $\widetilde{M_1}[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_k]$ is an EC for all $e'_1, \ldots, e'_k$. Therefore, by definition of ECs and assumption $\widetilde{M} = \widetilde{M_1}\,\widetilde{M_2}$, $\widetilde{M}[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_n]$ is an EC for all $e'_1, \ldots, e'_n$.

  Second, if for all $e_1, \ldots, e_k$ and $1 \leq i \leq k$, $\widetilde{M_1}[e_1, \ldots, e_{i-1}, [\,], e_{i+1}, \ldots, e_k]$ is *not* an EC then for all $e'_1, \ldots, e'_k$, $\widetilde{M_1}[e'_1, \ldots, e'_k] \in \mathit{Val}$, for otherwise there is no instantiation of $\widetilde{M} = \widetilde{M_1}\,\widetilde{M_2}$ that yields an EC, contradicting the assumption. Moreover, by assumption we have that $\widetilde{M_2}[e_{k+1}, \ldots, e_{i-1}, [], e_{i+1}, \ldots, e_n]$ is an EC, for some $e_{k+1}, \ldots, e_n$ and $k+1 \leq i \leq n$. By induction hypothesis and

$\widetilde{M} = \widetilde{M_1}\,\widetilde{M_2}$, there exists $k+1 \leq j \leq n$ such that $\widetilde{M}[e'_1, \ldots, e'_{j-1}, [\,], e'_{j+1}, \ldots, e'_n]$ is an EC for all $e'_1, \ldots, e'_n$.

- The case $\mathbf{exch}(\widetilde{M_1}, \widetilde{M_2})$ is similar.

The statement of the lemma follows from this result by an induction on the structure of (process) multicontexts M. □

A redex R in a multicontext M is an EC $D[E]$ or future EC $D[F]$ in M to which some reduction rule applies. If $\overline{e} = e_1, \ldots, e_n$ is a sequence and $I \in \{1, \ldots, n\}^*$ a sequence of indices of length m then we write $\overline{e}^I$ for the sequence $e_{i_1}, \ldots, e_{i_m}$ where $I = i_1, \ldots, i_m$. We write $[]_I$ for the sequence of hole markers $[]_{i_1}, \ldots, []_{i_m}$.

Lemma A.3 (Generalized context lemma for may-convergence) *For $n \geq 0$ and $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ possibly empty sequences of expressions:*

$$(\forall 1 \leq i \leq n : \; e_i \leq^{\mathsf{ev}}_{\downarrow} e'_i) \quad \Rightarrow \quad \forall M : M[e_1, \ldots, e_n]{\downarrow} \Longrightarrow M[e'_1, \ldots, e'_n]{\downarrow}$$

Proof. Let $M[e_1, \ldots, e_n]\downarrow$, so we have to show $M[e'_1, \ldots, e'_n]\downarrow$. We use induction on the following lexicographic ordering of pairs (l, n), where

(i) l is the length of a shortest succeeding sequence of reductions starting with $M[e_1, \ldots, e_n]$, and

(ii) n is the number of holes in M.

The claim holds for all pairs $(l, 0)$, since if M has no holes there is nothing to show. Now, let $(l, n) > (0, 0)$. For the induction step, we assume the claim holds for all pairs (l', n') that are strictly smaller than (l, n). We assume $\forall 1 \leq i \leq n : e_i \leq^{\mathsf{ev}}_{\downarrow} e'_i$ and let M be a multicontext with n holes such that $M[e_1, \ldots, e_n] \xrightarrow{l} p$ for some successful p. There are two cases:

- At least one hole of M is in EC position. Then let $1 \leq i \leq n$ be the position of the hole in EC position as determined by Lemma A.2. Let M_1 be the multicontext with $n-1$ holes defined by $M_1 \equiv M[[\,]_1, \ldots, [\,]_{i-1}, e_i, [\,]_{i+1}, \ldots, [\,]_n]$. Hence $M_1[e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n] \downarrow$ so that the induction hypothesis yields $M_1[e'_1, \ldots, e'_{i-1}, e'_{i+1}, \ldots, e'_n]\downarrow$. By Lemma A.2 and the choice of i there exists an EC $D'[E']$ such that $D'[E'] \equiv M[e'_1, \ldots, e'_{i-1}, [\,]_i, e'_{i+1}, \ldots, e'_n]$. Thus, $D'[E'[e_i]]\downarrow$ so that $D'[E'[e'_i]]\downarrow$ by assumption. The latter is $M[e'_1, \ldots, e'_n]\downarrow$.
- No hole of M is in EC position. Note that M may contain deep holes and also holes that are not in abstractions but also not in EC position. There are also no holes of the form $x\,\mathsf{c}\,[]$.
  If $l = 0$, then $M[e_1, \ldots, e_n]$ is successful, so $M[e'_1, \ldots, e'_n]$ is successful, too. Hence $M[e'_1, \ldots, e'_n]\downarrow$.

  If $l > 0$, then the first reduction step of $\xrightarrow{l}$ does also apply to $M[e'_1, \ldots, e'_n]$, where the same reduction at the same position is used, and in the case of FUT.DEREF(ev) and LAZY.TRIGGER(ev), the same future can be used in $M[e'_1, \ldots, e'_n] \xrightarrow{l} \ldots$.

  If the first reduction step of $\xrightarrow{l}$ is not a FUT.DEREF(ev), or it is a FUT.DEREF(ev) and the referenced value is an abstraction or variable, and there are no holes in

the referenced abstraction, then there is a multicontext M' with n holes, $M[\overline{e}] \to M'[\overline{e}^I] \xrightarrow{l-1} \ldots$, where I is a permutation of $\{1,\ldots,n\}$, and also $M[\overline{e}'] \to M'[\overline{e}'^I]$. We can apply the induction hypothesis to $M'[\overline{e}^I]$, since there is a reduction of length $l-1$ to a successful process.

If the reduction $\xrightarrow{l}$ begins with a FUT.DEREF(ev), the referenced value is an abstraction, and there are holes in the referenced abstraction, then more arguments are required. As a special notation for this part of the proof we indicate the occurrences of some of the expressions in the sequence $\overline{e}$ in the process in the context notation, which may lead to double square-bracket notation, but the meaning should be clear. The process expression $M[\overline{e}]$ is of the form $E[\overline{f_1}][x] \mid x \Leftarrow v[\overline{f_2}] \mid p[\overline{f_3}]$, where $\overline{f_j}$ are sequences of e_i's, such that $\overline{f_1}, \overline{f_2}, \overline{f_3} = \overline{e}$. Reduction of this process using FUT.DEREF(ev) results, before renaming, in $E[\overline{f_1}][v[\overline{f_2}]] \mid x \Leftarrow v[\overline{f_2}] \mid p[\overline{f_3}]$, and after a renaming within v in $E[\overline{f_1}][v_\rho[\rho\overline{f_2}]] \mid x \Leftarrow v[\overline{f_2}] \mid p[\overline{f_3}]$. The same reduction applied to $M[\overline{e}']$, where $\overline{f_1'}, \overline{f_2'}, \overline{f_3'} = \overline{e'}$ is $E[\overline{f_1'}][x] \mid x \Leftarrow v[\overline{f_2'}] \mid p[\overline{f_3'}] \xrightarrow{\text{FUT.DEREF(ev)}} E[\overline{f_1'}][v[\overline{f_2'}]] \mid x \Leftarrow v[\overline{f_2'}] \mid p[\overline{f_3'}]$. Note that we can in fact choose the renaming ρ such that it renames with variables that are fresh for *both* of the resulting processes. Therefore, after applying ρ to the second process, we obtain $E[\overline{f_1'}][v_\rho[\overline{\rho f_2'}]] \mid x \Leftarrow v[\overline{f_2'}] \mid p[\overline{f_3'}]$. We can apply the induction hypothesis to $E[\overline{f_1}][v_\rho[\overline{\rho f_2}]] \mid x \Leftarrow v[\overline{f_2}] \mid p[\overline{f_3}]$ and $E[\overline{f_1'}][v_\rho[\overline{\rho f_2'}]] \mid x \Leftarrow v[\overline{f_2'}] \mid p[\overline{f_3'}]$ using the sequences $\overline{f_1}, \overline{\rho f_2}, \overline{f_2}, \overline{f_3}$ and $\overline{f_1'}, \overline{\rho f_2'}, \overline{f_2'}, \overline{f_3'}$, since there is a reduction of length $l-1$ to a successful process, and an easy computation shows that the hypotheses $e_i\rho \leq^{\mathsf{ev}}_{\downarrow} e_i'\rho$ also hold for all parameters in $\overline{\rho f_2}$ and $\overline{\rho f_2'}$. □

Proposition A.4 (Context Lemma) *For all $e_1, e_2 \in Exp$:*

$$e_1 \leq^{\mathsf{ev}}_{\downarrow} e_2 \quad \textit{and} \quad e_1 \leq^{\mathsf{ev}}_{\Downarrow} e_2 \quad \Rightarrow \quad e_1 \leq e_2$$

Proof. The part for may-convergence follows from Proposition A.1. For the remaining part, we prove the claim that for all $n \geq 0$ and sequences $e_1, \ldots, e_n$, $e_1', \ldots, e_n'$,

$$\forall 1 \leq i \leq n : \; e_i \leq^{\mathsf{ev}}_{\downarrow} e_i' \;\wedge\; e_i \leq^{\mathsf{ev}}_{\Downarrow} e_i' \;\Rightarrow\; (\forall M : M[e_1', \ldots, e_n']\!\uparrow \Rightarrow M[e_1, \ldots, e_n]\!\uparrow)$$

The lemma then follows using the remarks on the relations on may- and must-convergence and divergence in Section 3. We prove the claim by induction on lexicographically ordered pairs (l, n) where

(i) l is the length of a shortest sequence of reductions starting with $M[e_1', \ldots, e_n']$ that ends in a process p with $p\Uparrow$, and

(ii) n is the number of holes in M.

If M has no holes there is nothing to show.
Now let $(l, n) > (0, 0)$. We analyze the two cases:

- At least one hole of M is in EC position. Then the same arguments as in the first part of the proof of Lemma A.3 show the claim.
- No hole of M is in EC position. If $l > 0$ then again the argumentation of part 2 of the proof of Lemma A.3 is used.
  The remaining case is $l = 0$, i.e., $M[e_1', \ldots, e_n']\Uparrow$. We have to show that

$M[e'_1,\ldots,e'_n]\Uparrow \Rightarrow M[e_1,\ldots,e_n]\uparrow$. Using the relations between may- and must-convergence and must- and may-divergence, respectively, stated in Section 3, an equivalent claim is $M[e_1,\ldots,e_n]\Downarrow \Rightarrow M[e'_1,\ldots,e'_n]\downarrow$. Using the precondition and Lemma A.3 we have $M[e_1,\ldots,e_n]\downarrow \Rightarrow M[e'_1,\ldots,e'_n]\downarrow$. Since obviously $M[e_1,\ldots,e_n]\Downarrow$ implies $M[e_1,\ldots,e_n]\downarrow$, the claim follows. □

B Incorrectness of Transformations

Let $I = \lambda x.x$. We prove Lemma 4.2 by giving counter-examples for every rule. For notational simplicity, we omit the ν-binders.

- THREAD.NEW(a): Let p_1 be the process $y \Leftarrow \lambda x.(\mathbf{thread}\, I)$. Thus, $p_1\Downarrow$ as it is already successful. Transforming it using THREAD.NEW(a) gives a process p_2, $y \Leftarrow \lambda x.z \mid z \Leftarrow (I\ z)$, which reduces to $y \Leftarrow \lambda x.z \mid z \Leftarrow u \mid u \Leftarrow z$, which is clearly must-divergent because of the cyclic subprocess, hence $p_2\uparrow$ follows.
- HANDLE.NEW(a): Let $y \Leftarrow \lambda x.\mathbf{handle}(\lambda u_1 \lambda u_2.u_2\,\mathbf{unit}) \mid x_1 \Leftarrow y\,\mathbf{unit} \mid x_2 \Leftarrow y\,\mathbf{unit}$ be the process p_1. Reduction of p_1 creates two handles and then terminates with a successful process, thus $p_1\Downarrow$. In the case it is transformed, we obtain the process $p_2 \equiv y \Leftarrow \lambda x.((\lambda u_1 \lambda u_2.u_2\,\mathbf{unit})\, y_1\, y_2) \mid y_2\,\mathsf{h}\,y_1 \mid x_1 \Leftarrow y\,\mathbf{unit} \mid x_2 \Leftarrow y\,\mathbf{unit}$ which will lead to a handle-error, i.e., p_2 is must-divergent.
- HANDLE.BIND(a): Consider the process $p_1 \equiv m \Leftarrow \lambda x.y\,\mathbf{unit} \mid n \Leftarrow y\,\mathbf{unit} \mid y\,\mathsf{h}\,z$ which is may- and must-convergent, whereas transformation results in the process $p_2 \equiv m \Leftarrow \lambda x.\mathbf{unit} \mid n \Leftarrow y\,\mathbf{unit} \mid y\,\mathsf{h}\,\bullet \mid z \Leftarrow \mathbf{unit}$. Note that p_2 is not successful but also not reducible. Hence p_2 is must-divergent.
- CELL.NEW(a): Applied within an abstraction, it is possible to share values, which are otherwise unshared, for instance let p_1, p_2 be the following processes:

$$
\begin{array}{lll}
p_1 \equiv & z \Leftarrow \lambda z.(\mathbf{cell}\, I) & \\
\quad \mid & x_1 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}),\mathbf{unit}))\,\mathbf{unit} & \\
\quad \mid & x_2 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}),\mathbf{unit}))\,\mathbf{unit} & \\
\end{array}
\qquad
\begin{array}{lll}
p_2 \equiv & z \Leftarrow \lambda z.w & \\
\quad \mid & w\,\mathsf{c}\,I & \\
\quad \mid & x_1 \Leftarrow (\mathbf{exch}((z\,\mathbf{unit}),\mathbf{unit}))\,\mathbf{unit} & \\
\quad \mid & x_2 \Leftarrow (\mathbf{exch}(((z\,\mathbf{unit}),\mathbf{unit}))\,\mathbf{unit} &
\end{array}
$$

  Process p_2 evolves from p_1 by applying CELL.NEW(a). We observe that $p_1\Downarrow$, since both **exch**-operations use their own cells and both will read the identity I. On the other hand, $p_2\Uparrow$ since the **exch**-operations use the same cell, so that the thread performing the second exchange remains stuck with an application of the form $\mathbf{unit}\,\mathbf{unit}$.
- CELL.EXCH(ev): The transformation is clearly not correct, since it can non-deterministically choose which exchange-operation to do first. The program $x \Leftarrow \mathbf{exch}(y,\mathbf{unit}) \mid z \Leftarrow \mathbf{exch}(y,\mathbf{unit}) \mid y\,\mathsf{c}\,x$ is may-convergent: after two reductions, the result is $x \Leftarrow \mathbf{unit} \mid z \Leftarrow x \mid y\,\mathsf{c}\,\mathbf{unit}$. Using the other possibility as transformation, a must-divergent program results: $x \Leftarrow x \mid z \Leftarrow \mathbf{exch}(y,\mathbf{unit}) \mid y\,\mathsf{c}\,\mathbf{unit}$.
- LAZY.NEW(a): This rule is not correct inside abstractions, since there may be a

sharing/desharing conflict. Let the processes p_1 and p_2 be defined as:

$$\begin{array}{lll} p_1 \equiv & y \,\mathsf{c}\, I & \\ & \mid w \Leftarrow \lambda x.\mathbf{lazy}(\lambda z.\mathbf{exch}(y,\mathbf{unit})) & \\ & \mid w_2 \Leftarrow (w\,\mathbf{unit})\,(w\,\mathbf{unit})\,\mathbf{unit} & \end{array} \qquad \begin{array}{lll} p_2 \equiv & y \,\mathsf{c}\, I & \\ & \mid w \Leftarrow \lambda x.w' & \\ & \mid w' \overset{susp}{\Longleftarrow} (\lambda z.\mathbf{exch}(y,\mathbf{unit}))\,w' & \\ & \mid w_2 \Leftarrow (w\,\mathbf{unit})\,(w\,\mathbf{unit})\,\mathbf{unit} & \end{array}$$

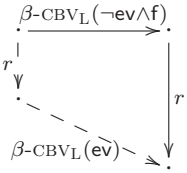
Applying LAZY.NEW(a) to p_1 results in process p_2. Note that $p_2 \Downarrow$, since only one **exch**-operation is performed (reading the identity I), whereas p_1 does not converge, since two exchange-operations are performed, and thus one of these results in **unit**.

- LAZY.TRIGGER(f): This transformation is not correct in arbitrary contexts, since it would force evaluation. An easy counterexample is $y \Leftarrow x \mid x \overset{susp}{\Longleftarrow} x$ which is convergent (it is successful), but becomes must-divergent after forcing the evaluation (because of the cyclic $x \Leftarrow x$).
- β-CBN: Let $p_1 \equiv y \Leftarrow \lambda x.\mathbf{unit}\ \bot$ and $p_2 \equiv y \Leftarrow \mathbf{unit}$, where $\bot$ is a must-divergent expression. Obviously p_2 is must-convergent while p_1 is must-divergent.

C Examples for the Forking and Commuting Diagrams

Lemmas 4.11, 4.15, and 4.20 follow by analyzing all overlappings of the corresponding transformation with reductions. In this section we give typical example cases for the non-trivial diagrams.

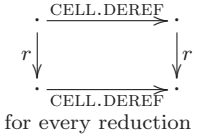
C.1 Diagrams for β-CBV$_{\mathrm{L}}(\neg\mathsf{ev}\wedge\mathsf{f})$



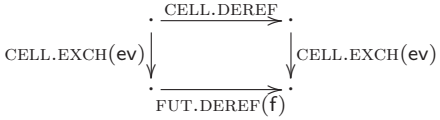
The non-trivial cases occur when an β-CBV$_{\mathrm{L}}(\neg\mathsf{ev}\wedge\mathsf{f})$ becomes a reduction β-CBV$_{\mathrm{L}}(\mathsf{ev})$, as expressed by the diagram. This case may occur if r is a LAZY.TRIGGER(ev) reduction, and if the redex of the transformation β-CBV$_{\mathrm{L}}(\neg\mathsf{ev}\wedge\mathsf{f})$ is inside a lazy future that gets triggered. Another case is that r is an β-CBV$_{\mathrm{L}}(\mathsf{ev})$ reduction, e.g.

$$\begin{array}{ccc} y \Leftarrow ((\lambda x.x)\ y)\ ((\lambda z.z)\ \mathbf{unit}) & \xrightarrow{\beta\text{-CBV}_{\mathrm{L}}(\neg\mathsf{ev}\wedge\mathsf{f})} & y \Leftarrow ((\lambda x.x)\ y)\ z \mid z \Leftarrow \mathbf{unit} \\ \Big\downarrow r & & \Big\downarrow r \\ y \Leftarrow x\ ((\lambda z.z)\ \mathbf{unit}) \mid x \Leftarrow y & \xrightarrow[\beta\text{-CBV}_{\mathrm{L}}(\mathsf{ev})]{} & y \Leftarrow x\ z \mid x \Leftarrow y \mid z \Leftarrow \mathbf{unit} \end{array}$$

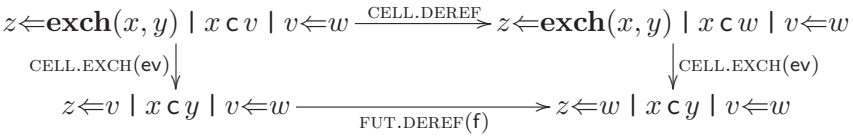
C.2 Diagrams for CELL.DEREF



for every reduction r

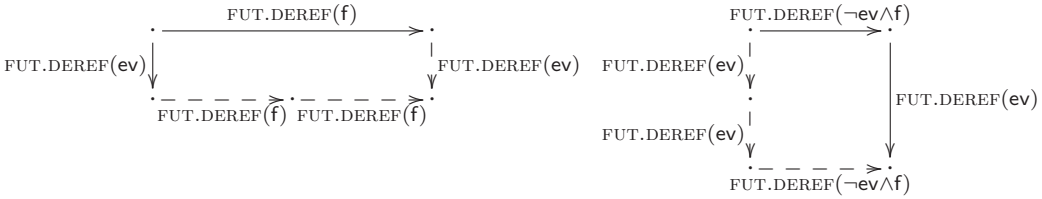


Cases for the first diagram are obvious. An example for the second diagram is:

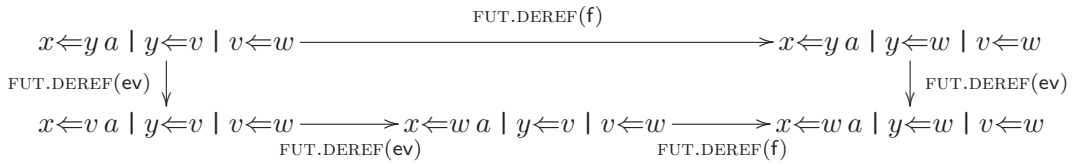


C.3 Diagrams for FUT.DEREF(f) *and* FUT.DEREF(¬ev∧f)

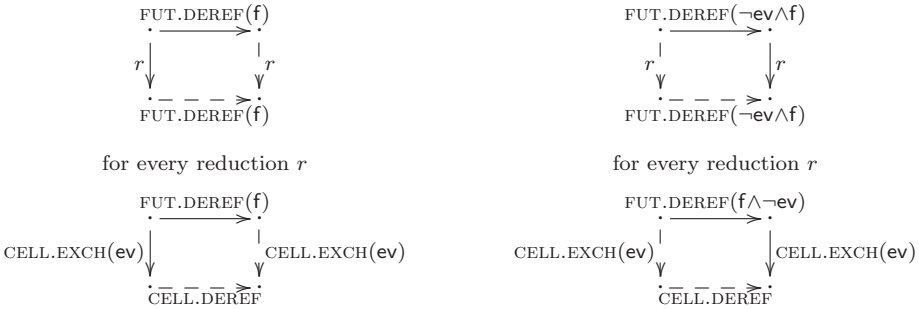
We show examples for the diagrams of Lemma 4.15.



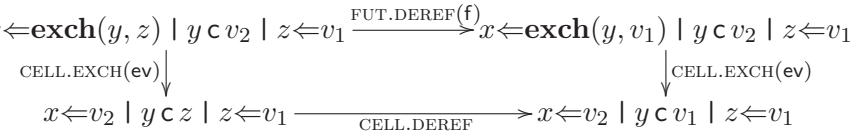
A typical example for both diagrams is:

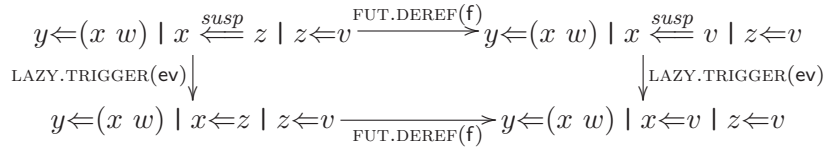


We now examine the following four diagrams:

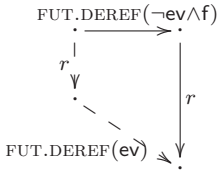


Beside simple commuting cases, where the transformation and the standard reduction do not influence each other, there are cases where the target of the dereferencing operation moves from a thread to a lazy thread or to a cell and vice versa. We show two examples:



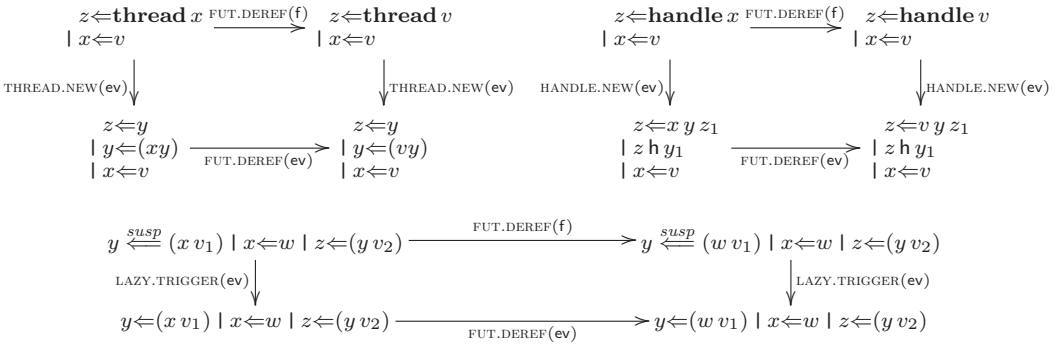


We now look at the diagram

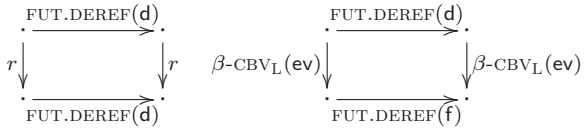


for $r \in \{\text{THREAD.NEW}(\mathsf{ev}), \text{HANDLE.NEW}(\mathsf{ev}), \text{LAZY.TRIGGER}(\mathsf{ev})\}$

We illustrate all three cases by examples:

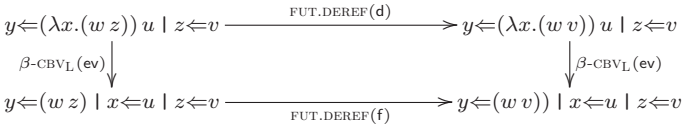


C.4 Diagrams for FUT.DEREF(d)

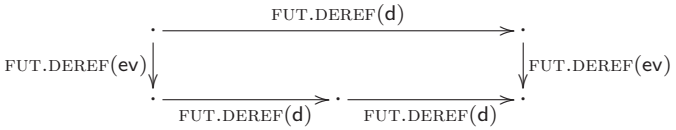


for every reduction r

The first diagram has the same special cases as the diagram for FUT.DEREF(f). The second diagram shows the only case where the target of a dereferencing operation is inside the body of an abstraction, but this is no longer the case after applying a standard reduction. An example for this case is:



The last diagram is:



An example for this case is: